Autonomous and Adaptive Systems

Introduction to TensorFlow and Keras

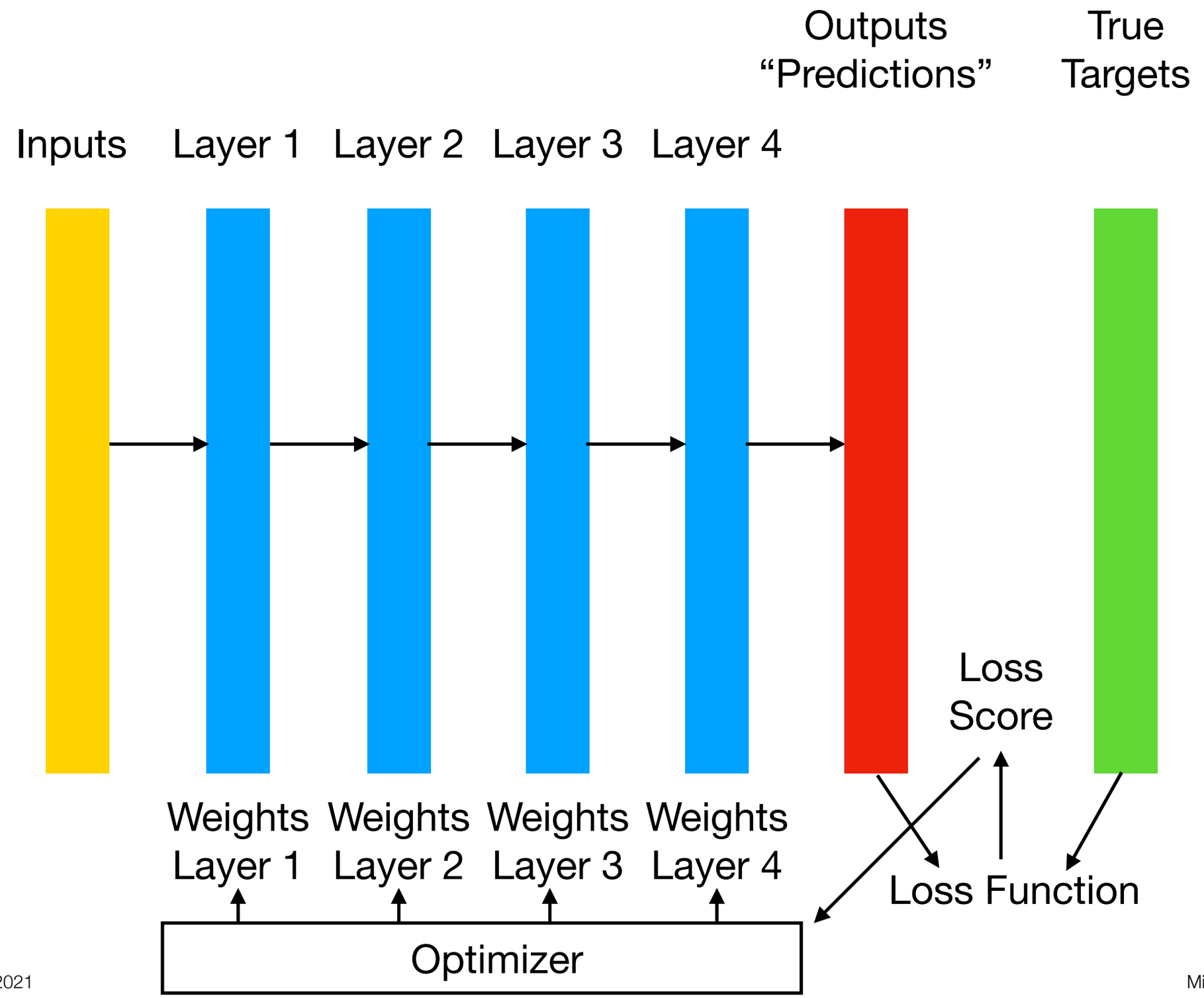
Mirco Musolesi

mircomusolesi@acm.org

TensorFlow and Keras

- TensorFlow is an end-to-end open source platform for machine learning.
- Current version is 2.0.
- It can be used with various high-level APIs like Keras.

Deep Neural Networks



Data Representations for Neural Networks

- Keras is based on the Numpy library.
- Numpy is one of the fundamental packages in Python for scientific computing.
- It contains:
  - efficient abstractions for N-dimensional arrays;
  - functions for managing N-dimensional arrays;
  - tools for integrating C/C++ and Fortran code;
  - linear algebra and Fourier transform operations;
  - random number generators.

Numpy Tensors

- Multi-dimensional arrays in Numpy are called *tensors*.
- Tensors are a sort of generalisation of the concept of matrix that you know.
- There are different types of tensors:
  - Scalars (0D tensors)
  - Vectors (1D tensors)
  - Matrices (2D tensors)
  - (3D) tensors
  - 4D+tensors

Scalars (0D Tensors)

- A tensor that contains only one number is called a scalar (or scalar tensors, or 0D tensor).
- In Numpy, a `float32` or `float64` number is a scalar tensor (or scalar array).
- The number of axes (also called dimension or rank) of a scalar tensor is 0 (obtained through `ndim`).

```
>>> import numpy as np

>>> x = np.array(256)

>>> x

array(12)

>>>x.ndim

0
```

Vectors (1D Tensors)

- An array of numbers is called a vector or 1D tensor.

```
>>> x = np.array([1, 2, 3, 4, 5])

>>> x

array([1, 2, 3, 4, 5])

>>> x.ndim

1
```

Matrices (2D Tensors)

- An array of vectors is a matrix, or 2D tensor. A matrix has two axes (rows and columns).

```
>>> x = np.array([1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 10, 11, 12])
>>> x.ndim
2
```

3D Tensors and Higher-Dimensional Tensors

- If you take an array of matrices, you obtain a 3D tensor.

```
>>> x = np.array([[[1, 2, 3, 4, 5],
                   [6, 7, 8, 9, 10],
                   [11, 12, 13, 14, 15]],
                  [[16, 17, 18, 19, 20],
                   [21, 22, 23, 24, 25],
                   [26, 27, 28, 29, 30]],
                  [[31, 32, 33, 34, 35],
                   [36, 37, 38, 39, 40],
                   [41, 42, 43, 44, 45]]])
>>> x.dim
3
```

- If you then take an array of 3D tensors, you obtain a 4D tensor.

- And if you take an array of 4D tensors, you obtain a 5D tensor and so on.

Key Attributes

- A tensor is defined by three key attributes:
  - *Number of axes (rank)*: a 3D tensor has three axes and a matrix of two axes. A batch of 2D images had 3 dimensions (more about batches soon).
  - *Shape*: this is a tuple of integers that describes how many dimensions the tensor has along each axis. For example, in the previous examples, the previous matrix example has shape `(3, 5)` and the 3D tensor example has shape `(3, 3, 5)`. A vector has a shape with a single element, such as `(5, )`, whereas a scalar has an empty shape `()`.
  - *Data type* (usually called `dtype` in Python). This is the type of the data contained in the tensor. For instance, a tensor's type could be `float32`, `uint8`, `float64` and so on.
    - No strings (because tensors are in pre-allocated contiguous memory segments).

Example: MNIST Dataset

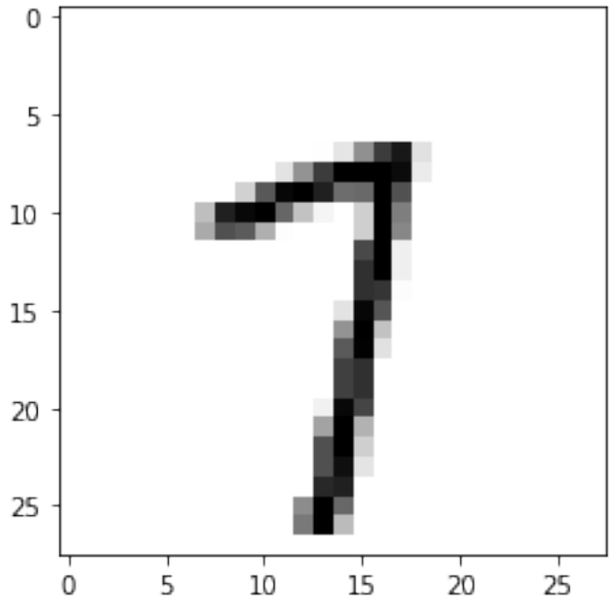
```
>>> from tensorflow.keras.dataset import mnist

(train_images, train labels), (test_images, test_labels) =
mnist.load_data()

>>> print (train_images).shape()

(60000, 28, 28)

>>> print(train_images.dtype())

uint8
```

We have a 3D tensor of 8-bit integers (60000 matrices of 28x28 integers). Each matrix is a greyscale image with coefficients between 0 and 255.

Example: MNIST Dataset

- Let's display the 42th digit in the 3D tensor using Matplotlib.

```
import matplotlib.pyplot as plt

import tensorflow

from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

digit = train_images[42]

plt.imshow(digit, cmap=plt.cm.binary)

plt.show()
```


0
5
10
15
20
25
0
5
10
15
20
25

Manipulating Tensors

- In the previous example, we selected a specific digit alongside the first axis using the syntax train_images[i]. Selecting specific elements in a tensor is called tensor slicing.

- The following example selects digits from 10 to 100 (with 100 not included) and put them in an array of shape (90, 28, 28).

```
>>> my_slice = train_images[10:100]

>>> print(my_slice.shape)

(90, 28, 28)
```

Data Batches

- In general the first axis (axis 0, because indexing starts at 0) in data sensors is the *sample axis*.

- In MNIST examples are images of digits.

- In deep learning we usually process batches. The following returns batches of MNIST digits with batch size of 128:

```
>>> batch = train_images[:128]

>>> batch = train_images[128:256]

>>> batch = train_images[128*n:128*(n+1)]
```

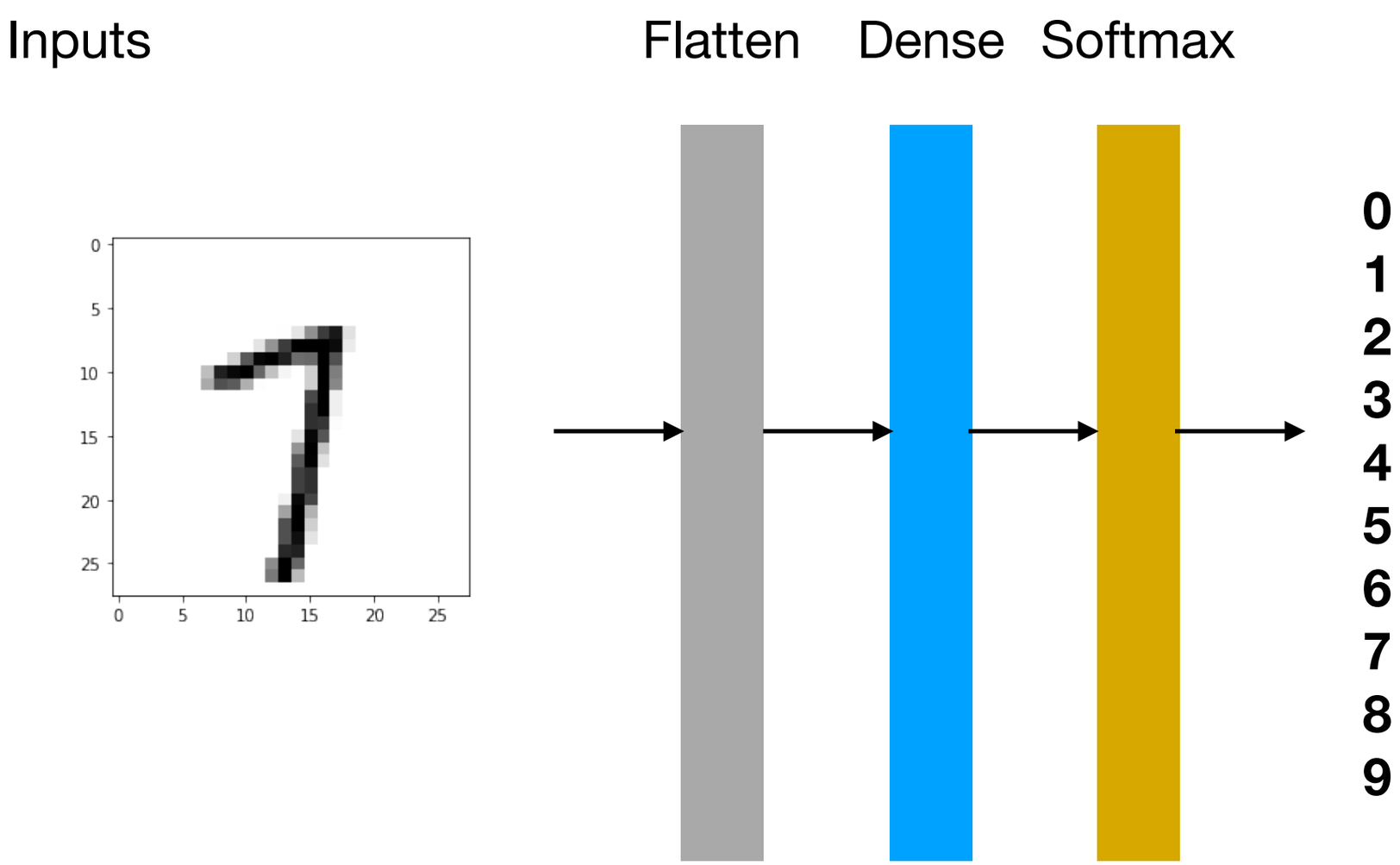
Key Components

- The key components of a network in Keras are:
  - *Layers*, which are combined in a network (model);
  - The *input data* and corresponding *targets*;
  - The *loss function*, which defines the feedback signal used for learning;
  - The *optimiser*, which determines how the the network is trained.

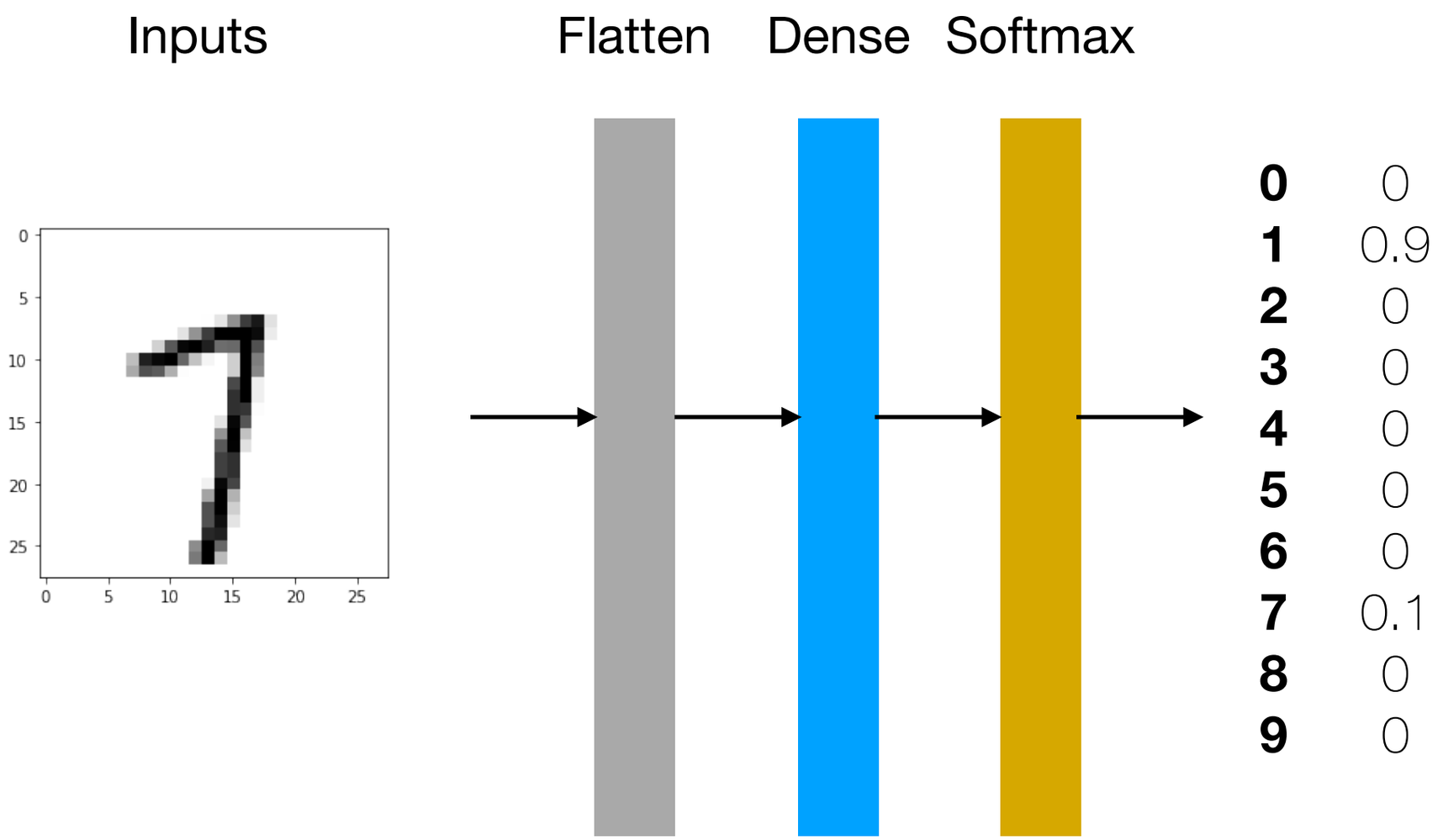
Example

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape = (28, 28)),
  tf.keras.layers.Dense(128, activation = 'relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation = 'softmax')
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = true)

model.compile(optimizer='adam',
              loss= loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

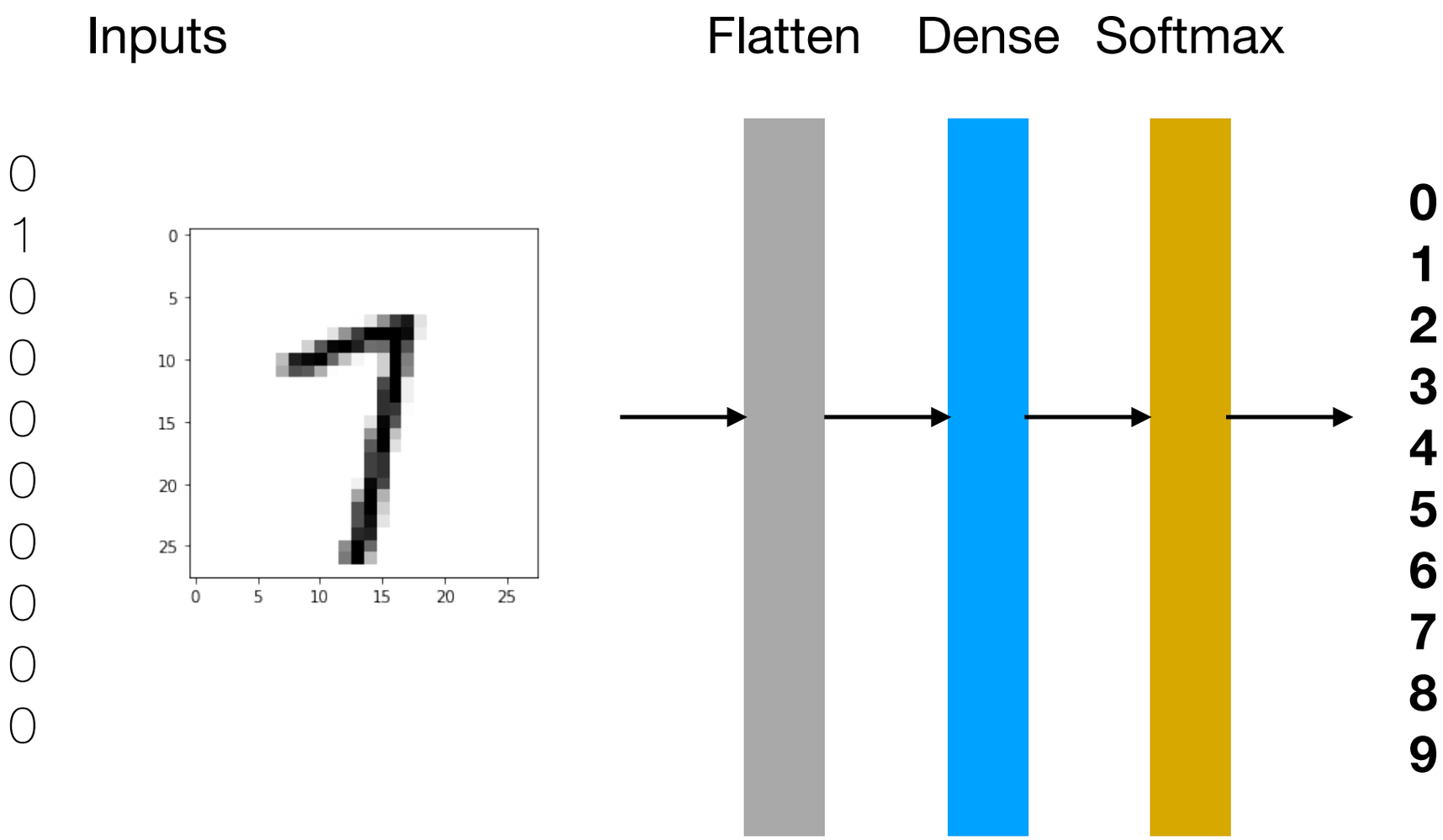
Example


Inputs
Flatten
Dense
Softmax
0
5
10
15
20
25
0
5
10
15
20
25
0
1
2
3
4
5
6
7
8
9

Example (after training)



Example of Pair Used for Training



Layers

- The fundamental data structure in neural networks is the *layer*.
- A layer is a data-processing module that takes in input one or more tensors and that outputs one or more tensors.
- Some layers are stateless, but more frequently layers have a state, i.e., the layer's *weights*.
- Different layers are appropriate for different tensor formats and different types of data.

Layers

- Different layers are appropriate for different tensor formats and different types of data.

- For instance simple vector data (2D tensors) are often processed by densely connected layers (`Dense` class in TensorFlow/Keras).

- Sequence data (3D tensors) are typically processed by recurrent layers (`LSTM` layer in TensorFlow/Keras).

Layers Compatibility

- Building deep learning Tensorflow/Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

- The notion of layer-compatibility here refers specifically to the fact the every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape

- Let us consider the following example:

```
from keras import layers


tf.keras.layers.Flatten(input_shape=(28, 28)),
```

- In this case we are creating a layer that will only accept 28 x 28 tensor that will flatten into a 784 vector.

Layers Compatibility

- Let us consider now:

```
tf.keras.layers.Dense(128, activation='relu')
```

- This layer can only be connected downstream to a layer that expects 128-dimensional vectors as input.

- We do not specify the input shape argument. The layer automatically infers its input shape as being the output shape of the layer that comes before.

.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = true)

model.compile(optimizer='adam',
              loss= loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Dropout

- Dropout is a regularisation techniques for avoiding overfitting in deep learning.

- It is based on removing (dropping) units in neural networks.

- Practically, it consists in randomly select a fraction of inputs and set them to 0 (0.2 in the example) at each update during training time.

Example

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = true)

model.compile(optimizer='adam',
              loss= loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Adam Optimizer

- Adam is a method for efficient stochastic optimisation of the weight which requires limited memory.
- In particular, the methods computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.
  - The first moment is the mean.
  - The second moment is the variance.
- The name Adam derives from *ada*ptive *m*oment estimation.
- It is particularly suited for the optimisation of stochastic objectives with high-dimensional spaces.

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma[*]
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba[*]
University of Toronto
jimmy@psi.utoronto.ca

ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

1 INTRODUCTION

Stochastic gradient-based optimization is of core practical importance in many fields of science and engineering. Many problems in these fields can be cast as the optimization of some scalar parameterized objective function requiring maximization or minimization with respect to its parameters. If the function is differentiable w.r.t. its parameters, gradient descent is a relatively efficient optimization method, since the computation of first-order partial derivatives w.r.t. all the parameters is of the same computational complexity as just evaluating the function. Often, objective functions are stochastic.

Other Optimizers

- There is a variety of optimizers, which implement a variety of algorithms that have been presented in the literature in the past years.
- You can also find a “vanilla” stochastic gradient descent called **`sgd`**.
- In **`sgd`** you can set the learning rate as follows:

```
optimiser=keras.optimizers(lr=0.1)
```

- You can find all the optimisers under **`keras.optimizers`**.

Example

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape = (28, 28)),
  tf.keras.layers.Dense(128, activation = 'relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation = ‘softmax')
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = true)

model.compile(optimizer ='adam',
              loss = loss_fn,
              metrics = ['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Loss Function: Sparse Categorical Cross Entropy

- It measures the distance between two probability distributions.
- In this case, it measures the distance between the probability distribution output by the network and the true distribution of the network.
- By minimising the distance between these two distributions you train the network to output a value that is as close as possible to the true value.
- The mathematical formula is as follows:

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)]$$

Other Losses

- We use the sparse categorical cross entropy loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case) and the classes are exclusive.

- Instead if we have one target probability per class for each instance (such as one-hot vectors, e.g., a vector with all zeros except for the corresponding class, i.e., 1 let's say in correspondence to 4 and 0 for the others), we will use categorical cross-entropy.

- In case of binary classification (with one or more binary labels) we will use the sigmoid (logistic) activation function instead of a softmax layer.

- You can find the documentation about the available losses under `tf.keras.losses`.

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology,

Example

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape = (28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = true)

model.compile(optimizer='adam',
              loss= loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Metric: Accuracy

- The accuracy function calculates how often predictions matches labels.

- It keeps two variables internally, one for the total account and one for the actual correct predictions.

Model Summary

- The model's **`summary()`** displays all the model's layers, including each layers name (which is automatically generated unless you set it).l its output shape and its non-trainable parameters.

- For example the first "dense layer" in our example has 784*128= 100352 parameters for the weights plus 128 parameters for the noise.

- Remember the formula

$$y_i = \sum_j w_j x_j + b_j$$ for each node each layer

x_j is a 784-dimensional vector over all the j (128 nodes) -> 128*784

```
>>> model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

Model Layers

- It is possible to access the layers and getting the values of the weights.
- This might be particularly important for debugging.
- The connection weights are usually initialised randomly and the biases to zero.
- Please note that you can set the initial values of the weights and the biases (see documentation for that, not needed in general).

Model Layer

```
>>> model.layers

[<tensorflow.python.keras.layers.core.Flatten at
0x1112f25d0>,
 <tensorflow.python.keras.layers.core.Dense at
0x65c386690>,
 <tensorflow.python.keras.layers.core.Dropout at
0x65c41bd10>,
 <tensorflow.python.keras.layers.core.Dense at
0x65c41bed0>]

>>> hidden1.name

‘dense’
```

Weights

```
>>> hidden1 = model.layers[1]
>>> weights, biases = hidden1.get_weights()
>>> weights

array([[ 0.0067068 ,  0.0444955 , -0.06906993, ...,  0.01917812,
         0.0571864 , -0.05091941],
       [ 0.07298902,  0.03879048, -0.07611574, ..., -0.0537806 ,
         0.04074715,  0.00973181],
       [ 0.01252006, -0.01403704, -0.0160231 , ...,  0.01923724,
         0.05250163,  0.01036092],
       ...,
       [ 0.02924076,  0.02288403, -0.05628259, ..., -0.00877253,
        -0.00582412,  0.04741878],
       [ 0.04770205,  0.00977071, -0.04665522, ...,  0.05703158,
        -0.0407013 ,  0.06358352],
       [ 0.06874896,  0.06481764, -0.07729561, ..., -0.03910512,
         0.07737378,  0.06048525]], dtype=float32)
```

Biases

```
>>> biases

array([ 0.15436447, -0.10200647, -0.21357839,  0.21006949, -0.03842947,
        0.12271099,  0.06399174,  0.08149339, -0.06964093,  0.01241389,
       -0.13083696,  0.01591714,  0.09653516,  0.11157246,  0.10728992,
        0.02496499,  0.04548947, -0.06404063,  0.00701183,  0.02384211,
        0.1734516 ,  0.02672073,  0.00857907,  0.00574968, -0.17148587,
       -0.03355813,  0.00542628,  0.0633577 ,  0.11311906,  0.08636814,
       -0.16180542,  0.01927859,  0.1280381 , -0.02355766,  0.04206759,
        0.08186771, -0.04619434,  0.13776259,  0.05937123, -0.1820551 ,
       -0.08898257,  0.01036496,  0.06565045,  0.02765695,  0.10485286,
       -0.13002615, -0.02703019,  0.01211506,  0.04039115,  0.09769704,
        0.02093072,  0.13398051, -0.12281404,  0.08231556, -0.01917284,
        0.04910273, -0.00876658,  0.08874512,  0.10410592, -0.00363306,
       -0.01080153,  0.12881082,  0.00631847,  0.1724271 , -0.02265261,
        0.21790975,  0.03310108, -0.11083294, -0.00176234, -0.20714733,
        0.014523  ,  0.16850016,  0.10065471,  0.22509257, -0.05203338,
        0.02265464,  0.01207699,  0.04100839,  0.0889826 ,  0.13444117,
        0.09707657,  0.22076793, -0.11211801,  0.09468664, -0.00847345,
        0.15046684,  0.09202639, -0.158566  , -0.12476195,  0.03112048,
        0.06387051, -0.00093352, -0.16877012, -0.0276502 , -0.0544424 ,
       -0.07698131,  0.05069286,  0.04499756,  0.11598568,  0.02794787,
        0.03360102,  0.03794497,  0.22851145, -0.08310112,  0.11295109,
        0.08821745,  0.02148618,  0.0684126 , -0.08671904,  0.08890733,
       -0.1351053 , -0.02953506, -0.02828003,  0.08103541,  0.01204334,
       -0.1303266 , -0.23609331, -0.04407614,  0.00159562,  0.13649496,
       -0.03989958, -0.03431396, -0.11035573,  0.03475453,  0.0170306 ,
       -0.05985961, -0.04696511,  0.22284533], dtype=float32)
```

Prediction

- We can use the predict() method of model to make predictions on new instances.
- For example, we can consider the digit in position 42 and print the corresponding probabilities:

```
>>> digit = x_test[42:43]

>>> y_prob = model.predict(digit)

>>> y_prob.round(2)


array([[0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]], dtype=float32)
```

- Alternatively we can predict the class directly associated to that output (numerical here, but there are methods to associate labels as well):

```
>>> y_pred = model.predict_classes(digit)

>>> print(y_pred.round(2))

4
```

Fine-Tuning Neural Network Hyperparameters

- Neural networks gives you great flexibility, but that is also their main drawback.
- You can use a variety of network architectures, you can change number of layers, number of neurons per layer, the type of activation function to use in each layer, the weight initialisation logic, etc.
- How do you select these parameters?
  - This is a search problem: a variety of methods are possible:
    - Grid search
    - Randomised search
    - …
  - Scikit-Learn offers a series of functions for grid search (`GridSearchCV`) and randomised search (`RandomizedSearchCV`). I would invite you to take a look at them.

Fine-Tuning Neural Network Hyperparameters

- A randomised search is not efficient. There are a variety of techniques and tools for optimising the parameters by “zooming” on certain ranges, etc.

- A variety of tools is available:

  - Hyperopt
  - Hyperas
  - Talos
  - Spearmint
  - …

- Companies also provide this optimisation as a service in the cloud (see Google Tuning).

Fine-Tuning Neural Network Hyperparameters

- Hyperparameter tuning is still an active area of research.
- Recent proposals also include evolutionary algorithms, i.e., algorithms that mimic biological systems based on reproduction, mutation, recombination and selection. Solutions are selected through a "survival" of the fitness in a population of potential candidates.
  - See Population Based Training (PBT) by DeepMind.

Population Based Training of Neural Networks

Max Jaderberg **Valentin Dalibard** **Simon Osindero** **Wojciech M. Czarnecki**

Jeff Donahue **Ali Razavi** **Oriol Vinyals** **Tim Green** **Iain Dunning**

Karen Simonyan **Chrisantha Fernando** **Koray Kavukcuoglu**

DeepMind, London, UK

Abstract

Neural networks dominate the modern machine learning landscape, but their training and success still suffer from sensitivity to empirical choices of hyperparameters such as model architecture, loss function, and optimisation algorithm. In this work we present *Population Based Training (PBT)*, a simple asynchronous optimisation algorithm which effectively utilises a fixed computational budget to jointly optimise a population of models and their hyperparameters to maximise performance. Importantly, PBT discovers a schedule of hyperparameter settings rather than following the generally sub-optimal strategy of trying to find a single fixed set to use for the whole course of training. With just a small modification to a typical distributed hyperparameter training framework, our method allows robust and reliable training of models. We demonstrate the effectiveness of PBT on deep reinforcement learning problems, showing faster wall-clock convergence and higher final performance of agents by optimising over a suite of hyperparameters. In addition, we show the same method can be applied to supervised learning for machine translation, where PBT

Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning

Felipe Petroski Such Vashisht Madhavan Edoardo Conti Joel Lehman Kenneth O. Stanley Jeff Clune

Uber AI Labs

{felipe.such, jeffclune}@uber.com

Abstract

Deep artificial neural networks (DNNs) are typically trained via gradient-based learning algorithms, namely backpropagation. Evolution strategies (ES) can rival backprop-based algorithms such as Q-learning and policy gradients on challenging deep reinforcement learning (RL) problems. However, ES can be considered a gradient-based algorithm because it performs stochastic gradient descent via an operation similar to a finite-difference approximation of the gradient. That raises the question of whether non-gradient-based evolutionary al-

1. Introduction

A recent trend in machine learning and AI research is that old algorithms work remarkably well when combined with sufficient computing resources and data. That has been the story for (1) backpropagation applied to deep neural networks in supervised learning tasks such as computer vision (Krizhevsky et al., 2012) and voice recognition (Seide et al., 2011), (2) backpropagation for deep neural networks combined with traditional reinforcement learning algorithms, such as Q-learning (Watkins & Dayan, 1992; Mnih et al., 2015) or policy gradient (PG) methods (Sehnke et al., 2010; Mnih et al., 2016), and (3) evolution strategies (ES) applied to reinforcement learning bench-

References

- Chapters 1-3 of Francois Chollet. Deep Learning with Python. Manning 2018.

- Chapter 10 of Aurelien Geron. Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow. Second Edition. O'Reilly 2019.

- TensorFlow Official Documentation website.